Swarming UAS II

MAJ Matthew Dabkowski, MS,
USMA, Department of Systems Engineering

Mr. James Cook
Donnelly & Moore Corporation

LTC Robert Kewley, PhD,
USMA, Department of Systems Engineering

May 5, 2010

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **05 MAY 2010**	2. REPORT TYPE	3. DATES COVERED **00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE **Swarming UAS II**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **USMA, Department of Systems Engineering,West Point,NY,10996**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**47**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Executive Summary

Problem Definition The Aviation and Missile Research, Development, and Engineering Center (AMRDEC) Advanced Science and Technology Unmanned Systems Office plans the way ahead for UAS. Their intent is to pursue technologies that increase the functionality of UAS while simultaneously reducing the workload on the user. Recent Operations Research Center of Excellence (ORCEN) research by MAJ Ed Teague, DSE-R-0808, showed that simple rule sets can organize multiple UAS to complete tasks with little or no input from controllers/users (Teague and Kewley, 2008). As a matter of design, DSE-R-0808 employed biomimicry to model a swarm of UAS as a colony of ants, where each UAS dynamically updates a global memory map, allowing pheromone-like communication (see DTIC: ADA489366 for additional details). In particular, the major findings of DSE-R-0808 were:

- Semi-Autonomous Self-Organizing (SASC) UAS are possible using preprogrammed tasks and pheromone communication methods.
- Depending on the rate of pheromone decay and the influence of distant pheromone levels, multiple UAS seem to sufficiently cover a large area without any input beyond a global rule set.
- SASC behaviors were tested via federated simulation, and its performance is comparable to preprogrammed routing without the overhead.

Technical Approach With this in mind, this year's objective was to advance the efforts of DSE-R-0808 through improved rule sets in order to achieve better results using doctrinal mission sets employing semi-autonomous, self-organizing UAS in dynamic environments. Accordingly, the principal research tasks were as follows:

- Develop improved rule sets for controlling swarming behavior.
- Develop a modeling and simulation test bed for swarming, small UASs in order to test the differential aspects of system components.
- Evaluate various UAS parameters to see how efficient/effective a swarm would be given a set of hardware (software) and recommend hardware (software) solutions.

Results In short, we successfully implemented and tested 9 different swarm controllers, using a standalone, custom Haskell simulation script we created. Moreover, in addition to several obvious performance metrics, we developed a novel measure, volume suppressed, which gauges the swarm's ability to minimize the *enemy's windows of opportunity* (space/time windows for the enemy to maneuver unobserved).

Disclaimer

Contents

"Restless thoughts, like a deadly swarm of hornets arm'd, no sooner found alone, but rush upon me thronging." – John Milton

1 Introduction

It was perfect. Sitting on the edge of the United States Military Academy's placid Lusk Reservoir, I enjoyed the warmth of a late spring sun, a pleasant easterly breeze, and the company of my children, as we passed the afternoon wetting a line. Like all previous fishing trips, the patience of my youngest was proportional to the frequency of the bite, and, although the weather was excellent, the fish were less than cooperative. Accordingly, he reeled in his line and decided to climb a small tree directly behind us. That's when I heard it - the guttural, animalistic scream that every parent dreads. My son was hurt.

As I simultaneously sprang up and swiveled my head to assess the situation, it took me a moment to register what I saw. There, swirling around my son like a chaotic yet strangely orchestrated cloud was a very angry, substantial swarm of hornets. I clenched my jaw, dropped my head, and went in. Needless to say, the insects carried the day, easily defeating a smarter, larger, and more advanced enemy; the swarm had won.

2 Background

2.1 Problem Description

While the concept of swarming is well known in nature and has been extensively researched, the dawn of the information age has only recently made such tactics possible for humans (Arquilla & Ronfeldt, 2000). Specifically, in their excellent RAND study titled "Swarming and the Future of Conflict," Arquilla and Ronfeldt remark:

> Swarming has two fundamental requirements. First, to be able to strike at an adversary from multiple directions, there must be large numbers of small units of maneuver that are tightly internetted—i.e., that can communicate and coordinate with each other at will, and are expected to do so. The second requirement is that the "swarm force" must not only engage in strike operations, but also form part of a "sensory organization," providing the surveillance and synoptic-level observations necessary to the creation and maintenance of "topsight" (Ibid, pg. 22)

With this in mind, bold advances in communication capabilities and network architectures satisfy the requirement for "tight internetting." Likewise, new data fusion software quickly transforms myriad, seemingly disparate pieces of information into collective intelligence, allowing "topsight." In short, we have the means; now all we need are the actors - the "the many and the small" (Ibid.).

While various manned, tactical combat formations and materiel could potentially fill this void, none show the promise of the Unmanned Aerial System (UAS). Capable of responsive intelligence gathering and offensive strikes with little or no risk of friendly casualties, UASs have firmly solidified their place in the military arsenals of many countries. For example, in 2008 the United States' Predator UASs alone saw a 94% increase in combat flight time from 2007, and over 1,000,000 UAS combat hours have been flown since the start of the Global War on Terror (See Figure 1) (Shachtman, 2009).[1] In order to meet this surging demand, the number of UASs has "increased from 167 unmanned planes to 5,331 in the past five years," [and] it's still not enough; . . . [d]emand for video is more than four times the supply" (Shachtman, 2008).

[1]Does not include man-portable systems (e.g., the Raven).

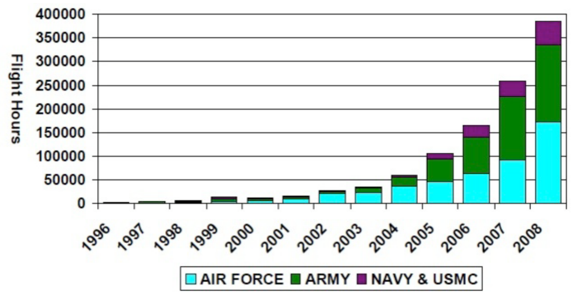


Figure 1: DoD UAS Flight Hours by Department by Fiscal Year

Coupled with the proliferation of UAS has been an ongoing and largely successful effort to miniaturize them. For instance, consider the Battlefield Air Targeting Micro Air Vehicle (BATMAV) depicted below.[2]

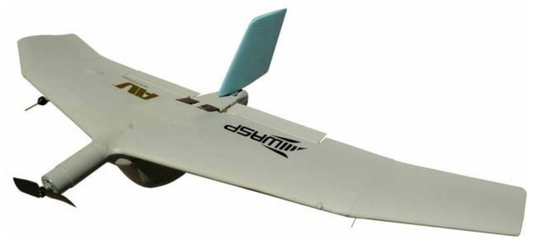


Figure 2: BATMAV UAS

Designed by AeroVironment for the United States Air Force's Special Operations, the BATMAV is 11.5" long, has a 16.5" wing span, and weighs only 1 pound (Department of Defense, 2009, pg. 68). Yet despite its diminutive size, the BATMAV is equipped with an "internal Global Positioning System / Inertial Navigation System, autopilot and two on-board cameras," and it can fly at roughly 40 mph for 45 minutes out to a range of 5 km (Ibid). While this is nothing short of astonishing, "future systems could fly transonically for 1,000 kilometers or endure for tens of hours" (Abatti, 2005, pg. 18-19).

Clearly, the age of the "many and the small" is dawning for UAS, carrying with it not only tactical opportunities but also dilemmas, not the least of which will be a shortage of operators. Specifically, current UASs, to include the BATMAV, require at least one operator per system and larger platforms typically require a team. If, however, the supply of UAS continues to grow in the face of increasing demand, multiple UAS will have to be controlled by a single operator. With the emerging n:1 UAS to operator issue on the horizon, it is not surprising that the second goal of Office of the Secretary of Defense's (OSD) FY2009-2034 Unmanned Systems Integrated Roadmap is to "[s]upport research and development activities to increase the level of automation in unmanned systems leading to appropriate levels of autonomy, as determined by the Warfighter for each specific platform" (Department of Defense, 2009). In short, algorithmic solutions are required.

2.2 Objective and Principal Tasks

The Aviation and Missile Research, Development, and Engineering Center (AMRDEC) Advanced Science and Technology Unmanned Systems Office plans the way ahead for UAS. Their intent is to pursue technologies that increase the functionality of UAS while simultaneously reducing the workload on the user. Recent Operations Research Center of Excellence (ORCEN) research by MAJ Ed Teague, DSE-R-0808, showed that simple rule sets can organize multiple UAS to complete tasks with little or no input from controllers/users (Teague & Kewley, 2008). As a matter of design, DSE-R-0808 employed biomimicry to model a swarm of UAS as a colony of ants, where each UAS dynamically updates a global memory map, allowing pheromone-like communication (see DTIC: ADA489366 for additional details). In particular, the major findings of DSE-R-0808 were:

- Semi-Autonomous Self-Organizing (SASC) UAS are possible using preprogrammed tasks and pheromone communication methods.

[2]Figure taken from the *Directory of U.S. Military Rockets and Missiles, Appendix 4: Undesignated Vehicles, Wasp* at http://www.designation-systems.net/dusrm/app4/wasp.html.

- Depending on the rate of pheromone decay and the influence of distant pheromone levels, multiple UAS seem to sufficiently cover a large area without any input beyond a global rule set.
- SASC behaviors were tested via federated simulation, and its performance is comparable to preprogrammed routing without the overhead.

With this in mind, this year's objective was to advance the efforts of DSE-R-0808 through improved rule sets in order to achieve better results using doctrinal mission sets employing semi-autonomous, self-organizing UAS in dynamic environments. Accordingly, the principal research tasks were as follows:

- Develop improved rule sets for controlling swarming behavior.
- Develop a modeling and simulation test bed for swarming, small UASs in order to test the differential aspects of system components.
- Evaluate various UAS parameters to see how efficient/effective a swarm would be given a set of hardware (software) and recommend hardware (software) solutions.

2.3 Literature Review

Controlling a swarm of UASs via digital pheromones is not new, and it is currently being pursued by multiple agencies. Most notably, in their 2002 article "Digital Pheromones for Autonomous Coordination of Swarming UAVs," Parunak et. al. describe their novel pheromone algorithm (ADAPTIV), which employs attractive and repulsive pheromones on a hex grid with roulette selection (Parunak *et al.*, 2002). More recently, Dasgupta details an agent-based approach in "A Multiagent Swarming System for Distributed Automatic Target Recognition Using Unmanned Aerial Vehicles" (Dasgupta, 2008). In addition to pheromones, Bae explores a chaos-based approach in "Target Searching Method in the Chaotic UAV," where UAS movement is simulated using Arnold's Equation (which models the behavior of noncompressive perfect fluids) and Chua's Equation (which models a simple electrical circuit) (Bae, 2004).

Beyond searching algorithms, Sujit et al. explore negotiation-based task allocation to multiple, neighboring UAVs (Sujit & Ghose, 2006), and efficient computation methods are discussed in Walter et al.'s "UAV Swarm Control: Calculating Digital Pheromone Fields with the GPU" (Walter *et al.*, 2006). This latter article is somewhat remarkable in that it demonstrates that pheromone field calculations can be performed 30 times faster on a computer's GPU versus CPU (Ibid.).

While incredibly valuable, however, the research to date has focused on detecting targets that are present, not on denying enemy action. Moreover, it has not incorporated a priori information about the underlying terrain or enemy activity to inform the swarm. Finally, although Bertuccelli and How provide an excellent, rigorous analysis of Bayesian updating for a single UAS (Bertuccelli & How, 2005), probabilistic updating has not been used to influence the behavior of the swarm.

2.4 Principal Assumptions

- **UAS are equipped with Traffic alert and Collision Avoidance Systems (TCAS)**. While airspace may be deconflicted using altitude, it is unreasonable to imagine that a large number of UAS (e.g., 100) operating in close proximity to one another would never cross paths. Additionally, as altitude increases the ability of sensors and communications hardware to interface effectively with the ground decreases.
- **UAS flight characteristics enable any adjacent grid to be visited next**. As seen in Figure 4, each UAS within the swarm will select its next location from one its eight adjacent grids. With this in mind, UAS of the future must be capable of rapid changes in direction, as opposed to the smooth, wide sweeping turns common in today's fixed wing UAS.

- **UAS and ground station communication and processing capabilities allow UAS to upload / download information to the global memory map in real-time**.
- **UAS are operating over a featureless, uninformed AO**. As discussed in Section 4, this assumption is important, because it forces the swarm to treat each grid square equivalently. In other words, locations on the ground cannot be distinguished as high or low threat.

3 Controllers

As seen in Figure 3, we establish a rectangular area of operations (AO) $\mathbb{A}$ consisting of $X \times Y$ square d-meter^2 grids, where X and Y represent the number of horizontal and vertical grid squares respectively, and the ordered pairs (x, y) represent their addresses, where $x \in [0, \ldots, X]$ and $y \in [0, \ldots, Y]$.

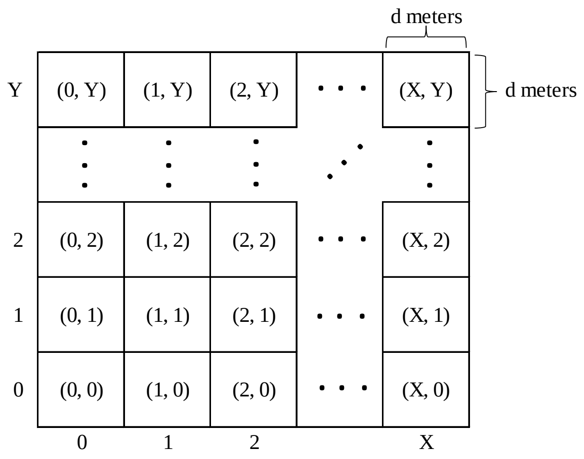


Figure 3: The Area of Operations

Controller #1 - Symmetric Random Walk

While the previous work of MAJ Ed Teague focused on pheromones and utilized a global memory map, the symmetric random walk does not utilize pheromones at all. As seen in Figure 4, at any time t, a UAS located at (x, y) randomly selects the next grid square to search from its set of 8 adjacent neighbors (or local neighborhood) with equal probability. In the event the UAS is located on the boundary of the AO, it simply drops any *out-of-bounds* grid squares from the set and redistributes the probability proportionally.

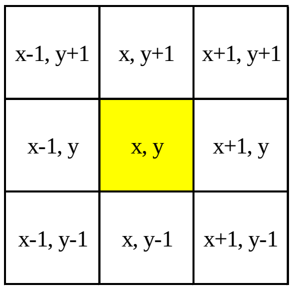


Figure 4: The Local Neighborhood

Controller #2 - Deterministic Local Neighborhood Search

In layman's terms, deterministic local neighborhood search directs UASs to focus their search on the neighboring grid square(s) with the greatest intervisitation time. Mathematically, at any time t, the weight of a given grid square (x, y) is given by $w_{x,y}(t) = p_{x,y}(t)$, where $p_{x,y}(t)$ represents the pheromone level in (x, y). At $t = 0$, $p_{x,y}(t) = 0 \; \forall x, y$, and each UAS within the AO searches its current grid square. At this point, exponential decay begins according to the relation $p_{x,y}(t) = \gamma_{x,y} e^{-\lambda(t - t_{x,y})}$ where λ is a positive, homogeneous decay constant, $t_{x,y}$ is the last time a UAS searched (x, y), and $\gamma_{x,y}$ is a binary parameter which equals 1 once (x, y) has been searched for the first time and 0 otherwise. In order to select the next grid square to search, a UAS located in (x, y) will examine the weights of the grid squares comprising its local neighborhood. Following examination, the UAS will search the neighboring grid square with the lowest weight, and ties are broken arbitrarily.

$$\overline{Q_k(t)} = \begin{cases} \overline{Q_1(t)} = \frac{\sum\limits_{i=0}^{x}\sum\limits_{j=y+1}^{Y} p_{i,j}}{x(Y-y)} & \text{for } \{(x-1,y+1),(x,y+1)\} \\ \overline{Q_2(t)} = \frac{\sum\limits_{i=x+1}^{X}\sum\limits_{j=y}^{Y} p_{i,j}}{(X-x)(Y-y+1)} & \text{for } \{(x+1,y+1),(x+1,y)\} \\ \overline{Q_3(t)} = \frac{\sum\limits_{i=x}^{X}\sum\limits_{j=0}^{y-1} p_{i,j}}{(X-x+1)(y-1)} & \text{for } \{(x,y-1),(x+1,y-1)\} \\ \overline{Q_4(t)} = \frac{\sum\limits_{i=0}^{x-1}\sum\limits_{j=0}^{y} p_{i,j}}{(x-1)y} & \text{for } \{(x-1,y),(x-1,y-1)\} \end{cases}$$

Controller #3 - Deterministic Local Neighborhood Search with Quadrant Averaging

In essence, quadrant averaging modifies the deterministic local neighborhood search algorithm to account for recent, global visitations, driving UASs to search the neighboring grid squares which (a) have not been visited recently and (b) lie in quadrants which have relatively large, recent average intervisitation times. Mathematically, at any time t, $w_{x,y}(t) = p_{x,y}(t) \times \overline{Q_k(t)}$, where $p_{x,y}(t)$ is defined as above and $\overline{Q_k(t)}$ represents the average pheromone concentration in quadrant $k = 1 \ldots 4$. As seen in Figure 5 on the following page, the quadrants are defined with respect to a UAS's current location (x, y), yielding the above equations for $\overline{Q_k(t)}$. As with Controller #2, the UAS will then evaluate the weights of the grid squares in its local neighborhood, and subsequently search the neighboring grid square with the lowest weight.

Controllers #4, 5, 6, and 7 - Mixture Models

Simply put, these algorithms randomly direct a UAS to use one of the first three controllers according to the probability mass function given below:

$$p(\text{UAS uses controller } c) = \begin{cases} p_1, & c = 1 \\ p_2, & c = 2 \\ p_3, & c = 3 \\ 0, & \text{otherwise} \end{cases}$$

For example, at any time t, there is p_2 probability that a given UAS will use Controller #2 to select its next grid square. In this way, Controllers #4 - #7 utilize Controllers #1 - #3 in proportion to the mixing parameters specified in Figure 6 below:

Controller Number	p_1	p_2	p_3
4	1 / 3	1 / 3	1 / 3
5	1 / 2	1 / 2	0
6	1 / 2	0	1 / 2
7	0	1 / 2	1 / 2

Figure 6: Mixing Parameters

Controllers #8 and 9 - Controllers #2 and 3 with Roulette Selection

Functionally, Controllers #8 and #9 are very similar to #2 and #3, respectively. In fact, the calculation of pheromone levels, exponential decay, and quadrant averaging are identical. However, after a UAS calculates the weights for its local neighborhood, it no longer automatically selects the adjacent grid square with the lowest weight – it merely prefers them through the application of *roulette selection*. Heuristically, roulette selection generates a

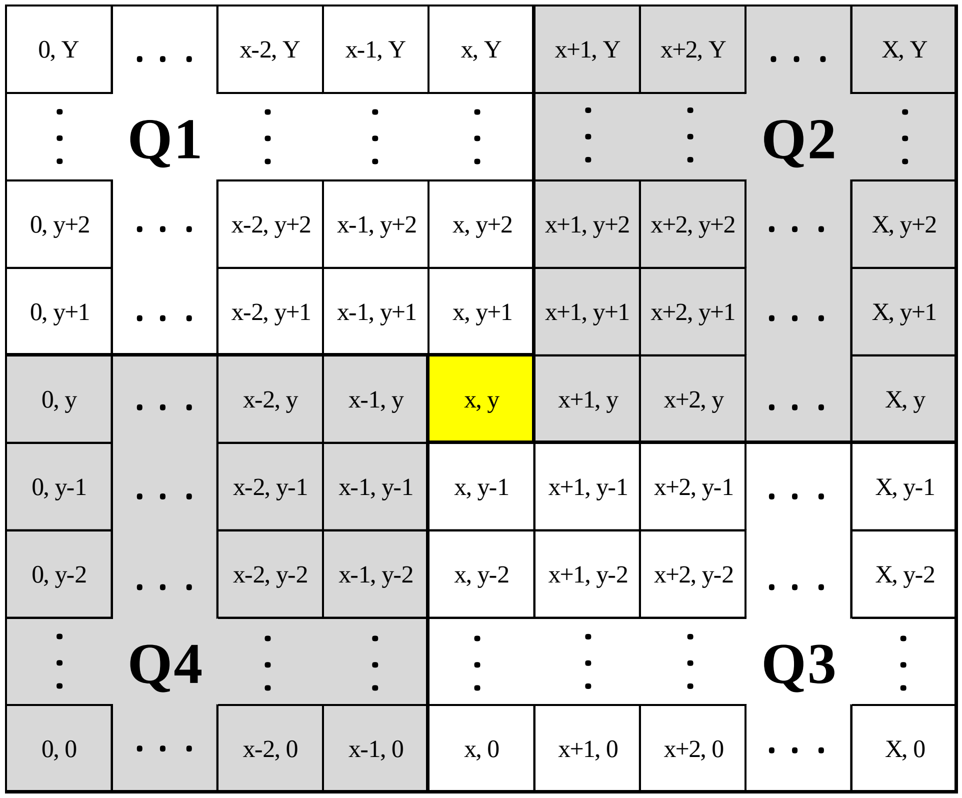


Figure 5: The Quadrants

probability mass function for a UAS according to the weights of its local neighborhood. Akin to a roulette wheel, the UAS then determines where a uniformly distributed random variable between 0 and 1 (e.g., the ball) falls within the cumulative distribution function (e.g., the wheel's bins; see Figure 7 below).

Local Neighbor of (x,y)	$w_{x,y}$	Normalized $w_{x,y}$
x-1, y+1	0.423	0.098
x, y+1	0.954	0.222
x+1, y+1	0.769	0.179
x-1, y	0.019	0.004
x+1, y	0.212	0.049
x-1, y-1	0.408	0.095
x, y-1	0.938	0.218
x+1, y-1	0.577	0.134
Total	4.301	1.000

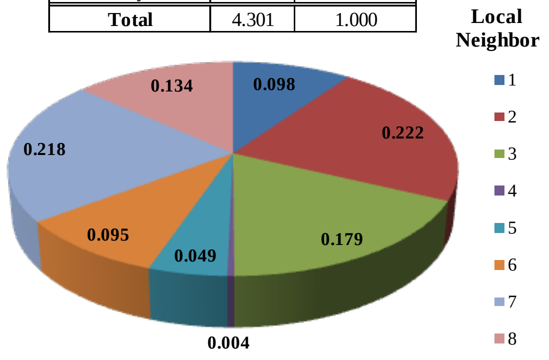


Figure 7: Roulette Selection Example

In this way, the UAS is more likely to select adjacent grid squares with lower weights, thereby injecting a stochastic element into an otherwise deterministic algorithm. Interestingly enough, the use of roulette selection in the control of swarming UAS is not new. In particular, in their paper "Digital Pheromones For Autonomous Coordination Of Swarming UAS" Parunak et al. describe its use in the ADAPTIV algorithm (Parunak *et al.*, 2002).

4 Initial Metrics

Developing improved rule sets implies that we want or need to (a) fix a problem with our current rule set, (2) ramp-up its current performance, or (3) inject additional capabilities. Regardless of the reason, each requires that we develop a set of metrics to assess and compare the algorithms. Of course, this begs the question, how does an effective swarm behave? In order to address this, we assume that the UAS are operating on a featureless, uninformed AO. Put another way, we do not have intelligence (enemy or geospatial) about the AO. This distinction is quite important, because it forces us to treat each grid square equivalently. In other words, we have no reason to suspect that any given grid square is more or less likely to contain a target. Under this assumption, some potential measures of swarm effectiveness are:

- The coverage must be complete (As t $\rightarrow \infty$, all cells are visited.)
- The coverage must eliminate sanctuaries
  - Geospatially: All cells are recurrent.
  - Temporally: The coverage minimizes the time between cell visitation.
- The coverage should be uniform (As t $\rightarrow \infty$, the number cell visitations per cell are nearly equivalent.)
- The behavior should appear random (acyclic geospatially and temporally)

Accordingly, we developed the following metrics:

1. **Standard Deviation of Visitation Counts** ($\sigma_{N_{x,y}}$): $N_{x,y}$ is defined as the number times (x, y) is visited (observed) by a UAS during a simulation run. On a featureless, uninformed AO, a good algorithm should evenly distribute its visitations between the grid squares. Accordingly, we can use the standard deviation of $N_{x,y}$ to gauge the closeness of the visitation counts, where smaller $\sigma_{N_{x,y}}$ are preferred.
2. **Maximum Initial Visitation Time** ($\max\limits_{\mathbb{A}} t_{x,y_{(1)}}$): Max $t_{x,y_{(1)}}$ is defined as the latest time a UAS visits any (x, y) during a simulation run. One may think of this quantity as a

measure of the diffusive speed for the swarm, where good performing algorithms visit each grid square within the AO as soon as possible, allowing for the quick detection of static targets that exist at $t = 0$.

3. **Maximum Maximum Intervisitation Time** ($(\underset{\mathbb{A}}{\text{Max}}(\text{Max } \triangle t_{x,y}))$: Mathematically, the times between UAS visitations for (x, y) or $\triangle t_{x,y}$ are defined by the sequence $\{t_{x,y_{(2)}} - t_{x,y_{(1)}}, t_{x,y_{(3)}} - t_{x,y_{(2)}}, \ldots, t_{x,y_{(n+1)}} - t_{x,y_{(n)}}\}$ for $n \in [0, \ldots, N_{x,y} - 1]$. Clearly, the maximum of this sequence represents the best opportunity for an enemy combatant to act within (x, y) without being observed. Accordingly, good algorithms should minimize this quantity across the entire AO.

4. **Maximum Average Intervisitation Time** ($\underset{\mathbb{A}}{\text{Max}}\overline{\triangle t_{x,y}}$): $\overline{\triangle t_{x,y}}$ is defined as the average time between UAS visitations for (x, y). Similar to the above, by minimizing the maximum average intervisitation time, a swarming algorithm can effectively frustrate the enemy's uninhibited use of any portion of the AO.

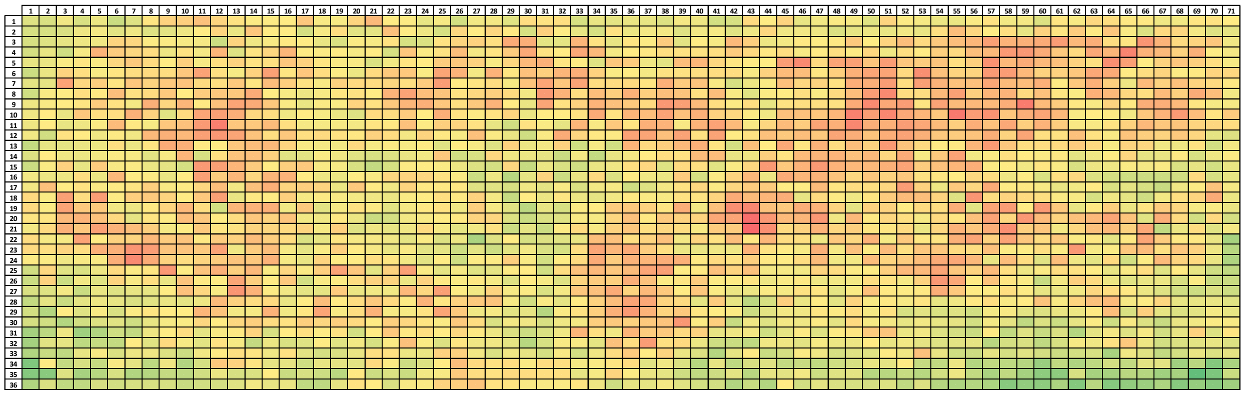
5 Preliminary Experimentation and Analysis

Prior to the development of the 8 new controllers, preliminary testing and evaluation were performed on last year's algorithm (Controller #3). Specifically, using a swarm of 10 UAS ($S = 10$), an arbitrarily sized AO ($X = 71, Y = 36$), and a sufficiently long run time ($T = 30,000$), we calculated the relative and absolute performance of Controller #3 on the four previously mentioned metrics. Additionally, in order to provide informative, realistic results, it was necessary to scale the dimensions of each cell in light of the published cruising speed and future sensor capabilities of the Shadow UAS.

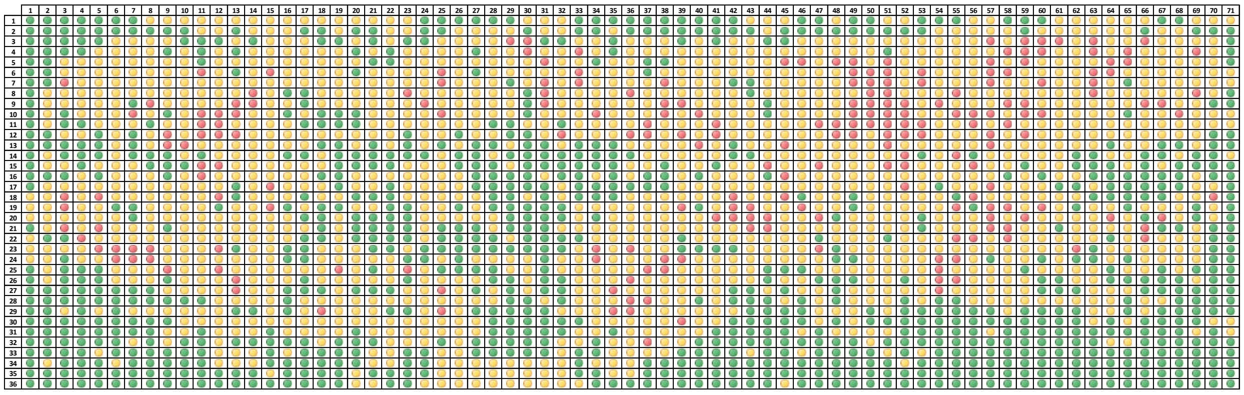
For example, according to FMI 3-04.155, the Shadow's estimated, unclassified cruising speed is 70 knots (Department of the Army, 2006). Following simple dimensional analysis, this equates to 35.98 meters / second. Moreover, in Table 2-7 of FMI 3-04.155 it states that the Shadow's current observable target area is 3.5m × 3.5m (Ibid., pg. 2-9). If we assume an order-of-magnitude increase in the width of the observable target area by FY2020, then we can conveniently set the length and width of each grid square to 35m. Using this scaling, we can now equate each unit of simulated time to 1 second, providing a good approximation for the amount of time required to transit and observe a given cell within our AO. Finally, in order to initiate the replications, we directed the swarm to enter the AO from the north at cell $(20, 1)$ in an echelon-left formation.

5.1 Visitation Counts

Throughout the simulation, the by-cell visitation times were continuously calculated, updated, and stored in an output array. This data was then post-processed using simple Excel pivot tables and conditional formatting to generate the visitation counts figures seen below. As we examine the relative performance of Controller #3, we notice a greater concentration of red-shaded cells in the center of the AO, while the boundaries appear to contain more green. Moreover, when the data is examined in an absolute sense, where cells visited more than 100 times are shaded green, the affinity of the UAS for the boundary becomes more apparent. Finally, it is worth noting that minimum and maximum $N_{x,y}$ were 48 and 176 respectively, and $\sigma_{N_{x,y}} = 16$.



Relative Performance
(Green is highest; Red is lowest)

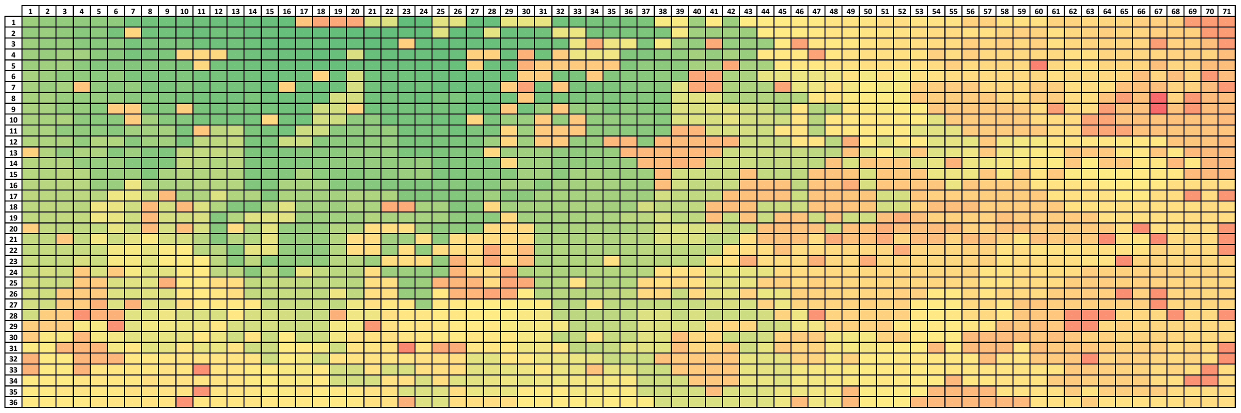


Absolute Performance
(Green $\geq$ 100 visits $>$ Yellow $\geq$ 75 visits $>$ Red)

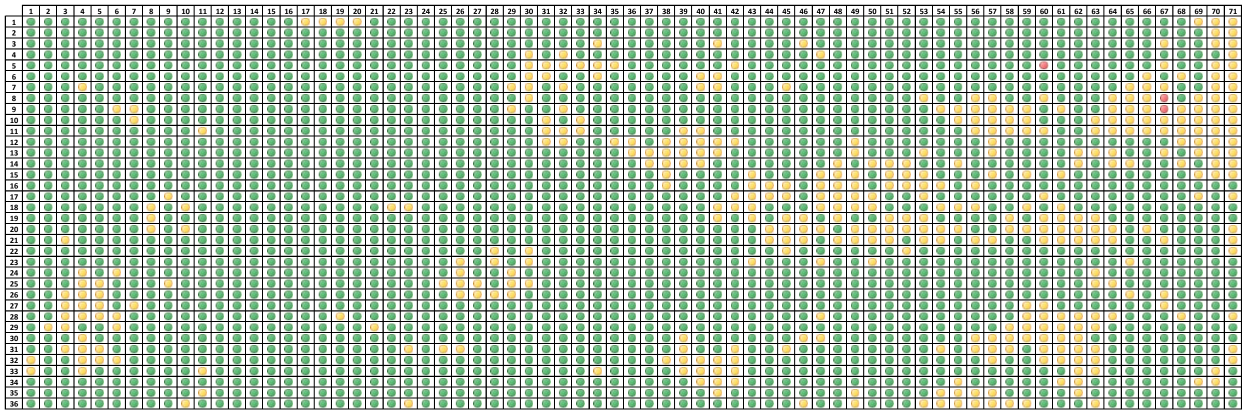
Figure 8: Cell Visitation Counts for Controller #3

5.2 Initial Visitation Times

As seen in the relative performance graph below, there is a clear bias for the area surrounding the swarm's point of entry. This is to be expected, and it serves as a form of verification more than an indicator of performance. However, when the same data is examined in absolute terms, where the maximum initial visitation time is 735 seconds and all cells visited in less than 5 minutes are shaded green, the overall *initial diffusive performance* seems adequate.



Relative Performance
(Green is earliest; Red is latest)

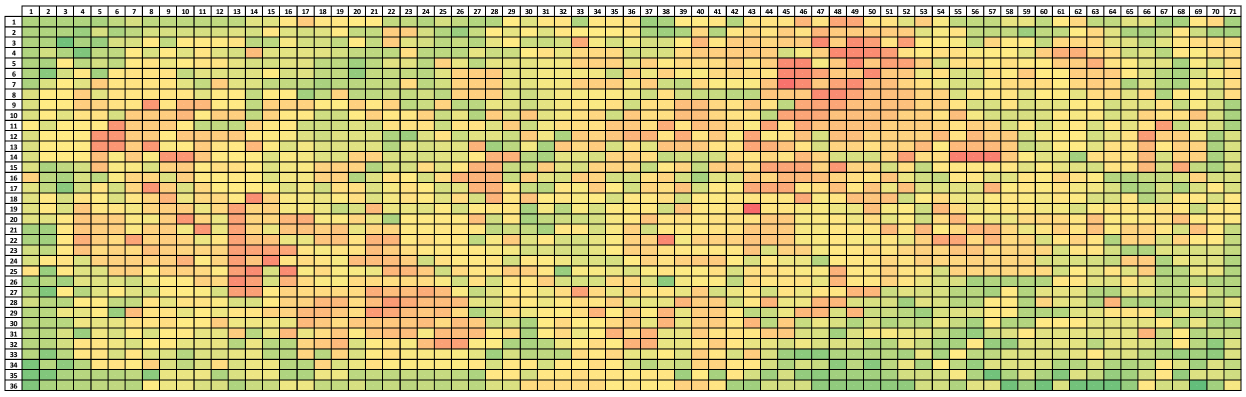


Absolute Performance
(Red $\geq$ 10 minutes $>$ Yellow $\geq$ 5 minutes $>$ Green)

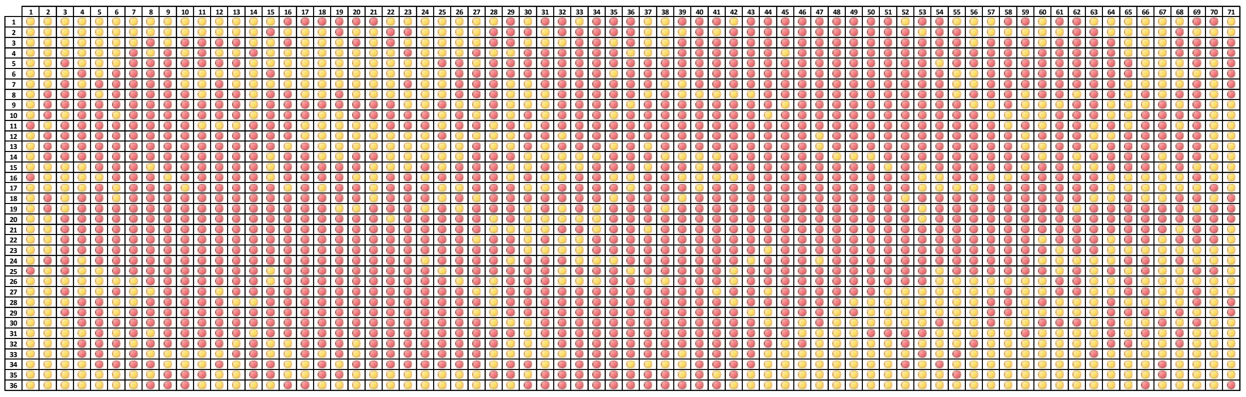
Figure 9: Maximum Initial Visitation Times for Controller #3

5.3 Maximum Intervisitation Times

While the initial visitation times seem quite acceptable, the maximum intervisitation times are worrisome. Specifically, in relative terms, it appears that the UAS have an affinity for the boundary, as shown by the denser concentration of green cells around the edges of the AO. Moreover, when we set the minimum acceptable performance to 1 hour, nearly all cells in the center of the AO fail (as indicated by the red shaded cells). In fact, the worst performing cell had a maximum intervisitation time of 7032 seconds (or 1 hour and 57 minutes)! When we contrast this poor performance with the acceptable performance on the initial visitation times (where the swarm visits each cell in roughly 10 minutes), one thing is clear – diffusive performance is not constant.



Relative Performance
(Green is shortest; Red is longest)

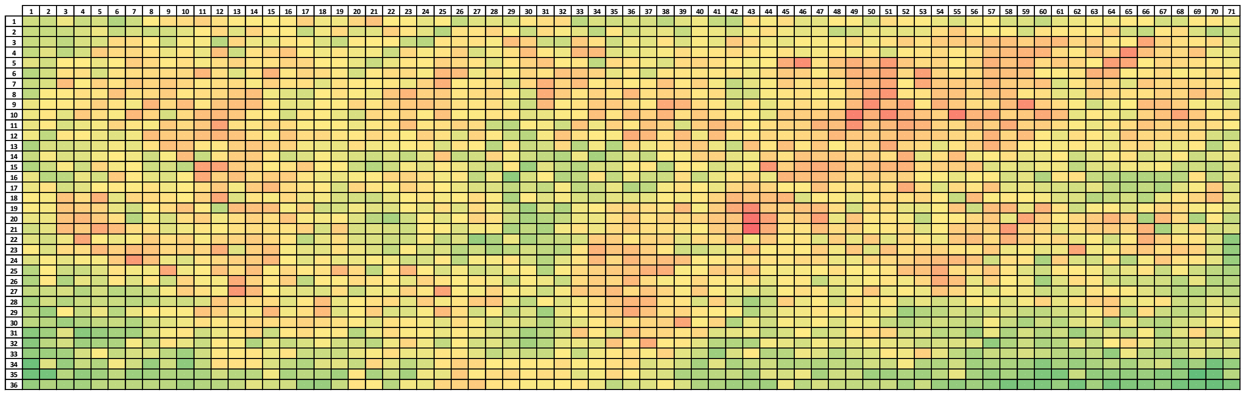


Absolute Performance
(Red $\geq$ 1 hour $>$ Yellow $\geq$ 30 minutes $>$ Green)

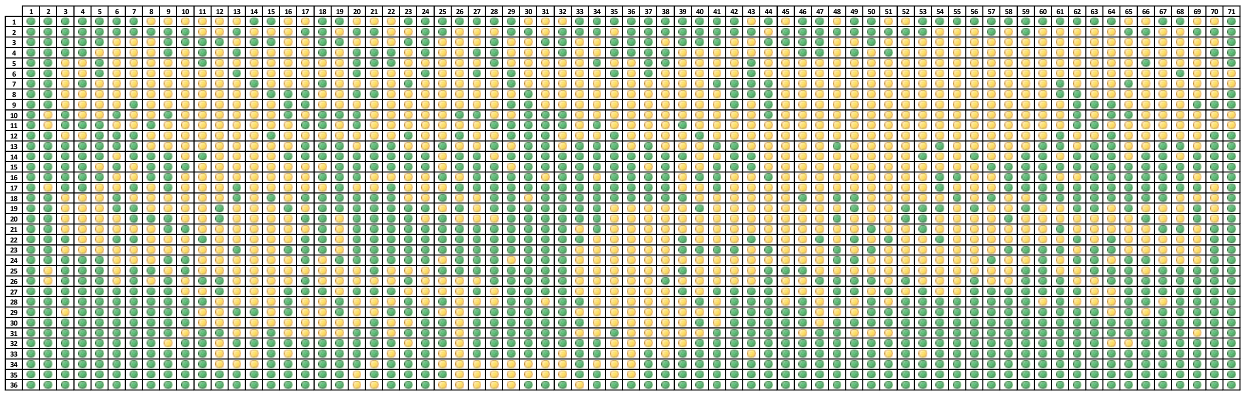
Figure 10: Maximum Intervisitation Times for Controller #3

5.4 Average Intervisitation Times

Similar to the maximum intervisitation times, the relative, average intervisitation times indicate that the UAS have an affinity for the boundary. To the contrary, however, the absolute performance also appears acceptable, as the entire AO is shaded green and yellow, indicating that all cells are visited in less than 10 minutes on average. Therefore, when we consider (a) that the maximum intervisitation times are unacceptable and (b) that the initial and average diffusive performance seem adequate, it appears that the swarm's entropy decreases as a function of time. In other words, the longer the simulation runs, the more ordered the behavior of the swarm becomes.



Relative Performance
(Green is shortest; Red is longest)



Absolute Performance
(Red $\geq$ 10 minutes $>$ Yellow $\geq$ 5 minutes $>$ Green)

Figure 11: Average Intervisitation Times for Controller #3

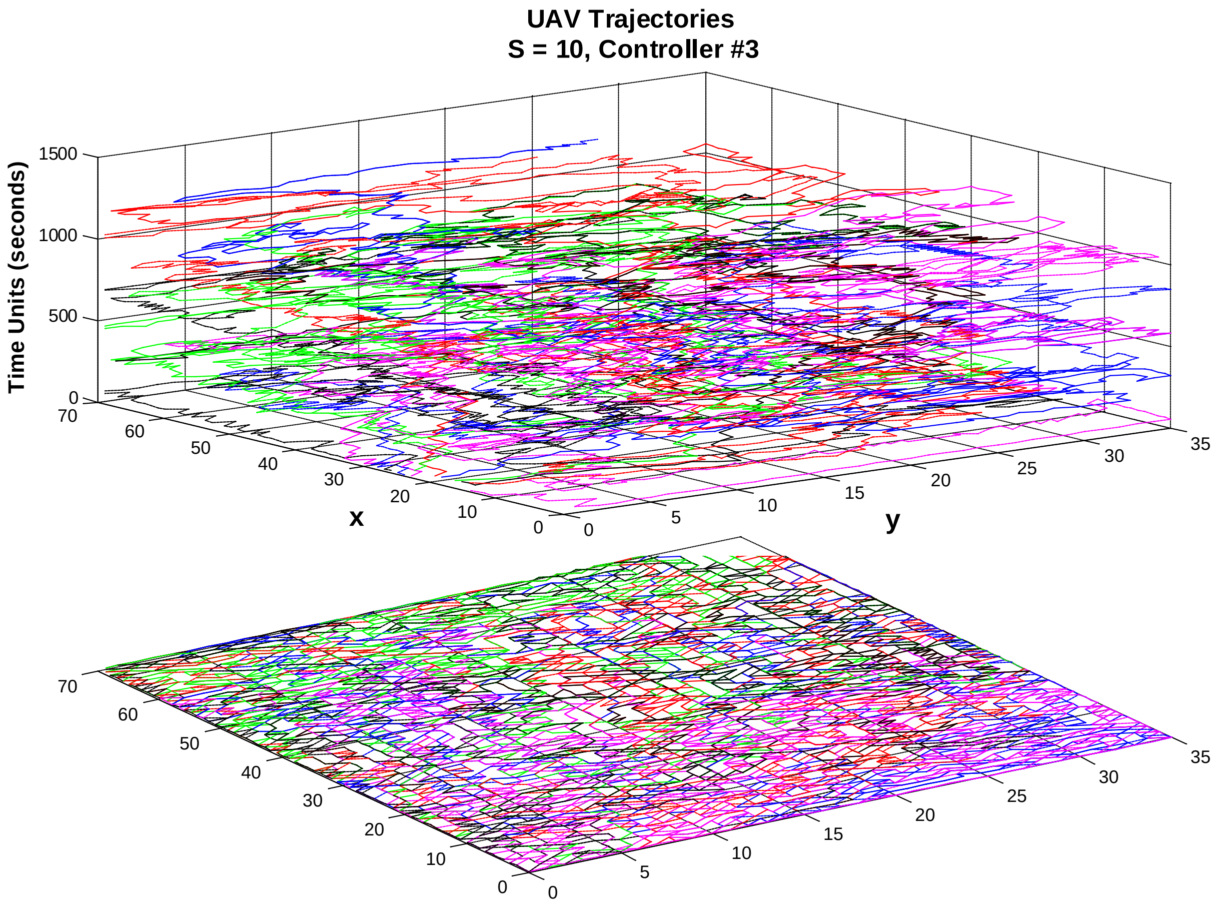


Figure 12: UAV Trajectories in Space/Time (t = 1 . . . 1000)

5.5 UAV Trajectories in Space and Time

In order to test the *decreasing entropy* hypothesis, we plotted the trajectories of the UAS in space and time. Seen Figure 12 below, the horizontal plane represents the AO, and the vertical dimension is time. Within this volume each of the erratic, multicolored lines represents the location of a UAS. In this way, Figure 12 provides a snapshot of the swarm's historical contrails over the first 1000 units of simulated time. When viewed from the side (the top panel) or from the top (the bottom panel) the swarm's behavior appears Brownian, and it fills the volume. In other words, there is no apparent order; it appears to be operating as intended.

When we advance the simulation clock to 22700 and plot the swarm's contrails for the subsequent 1000 time units, a much different picture emerges. Seen in Figure 13, the volume is not filled, and the coverage appears sparse. In fact, the swarm appears to have actually lost UAS, as many of the colorful lines from Figure 12 appear to missing! Upon closer inspection, however, we are actually observing the output of a very interesting development within the swarm – flocking. Evident in the small, left inset of Figure 13, the black line is actually a tight grouping of UAS flying together, confirming the the *decreasing entropy* hypothesis.

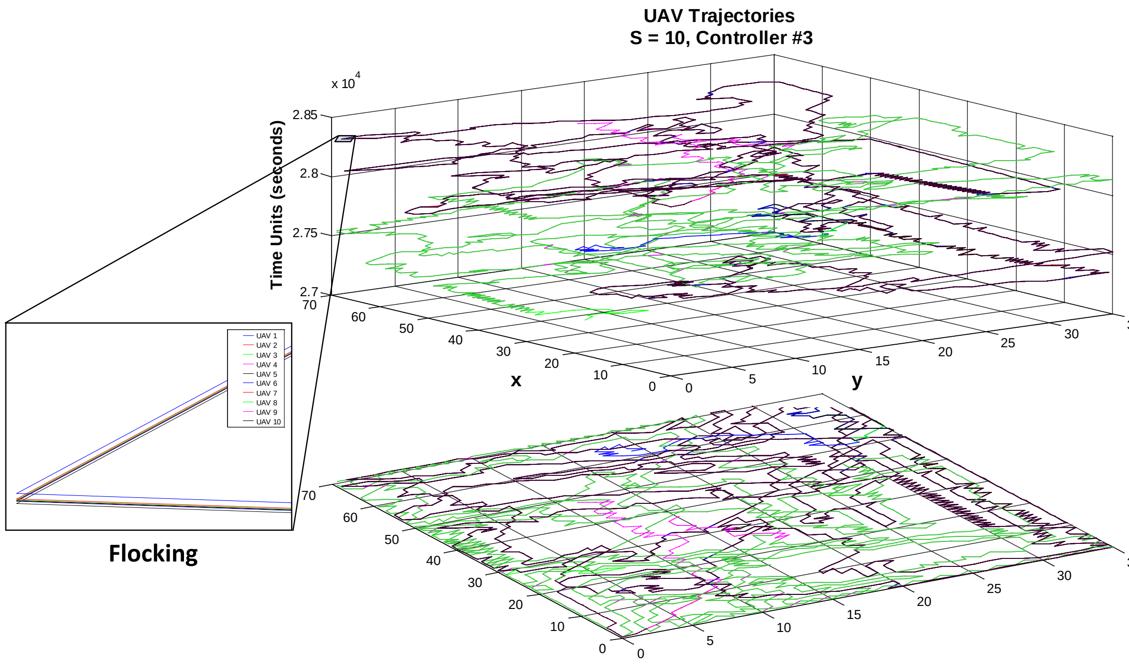
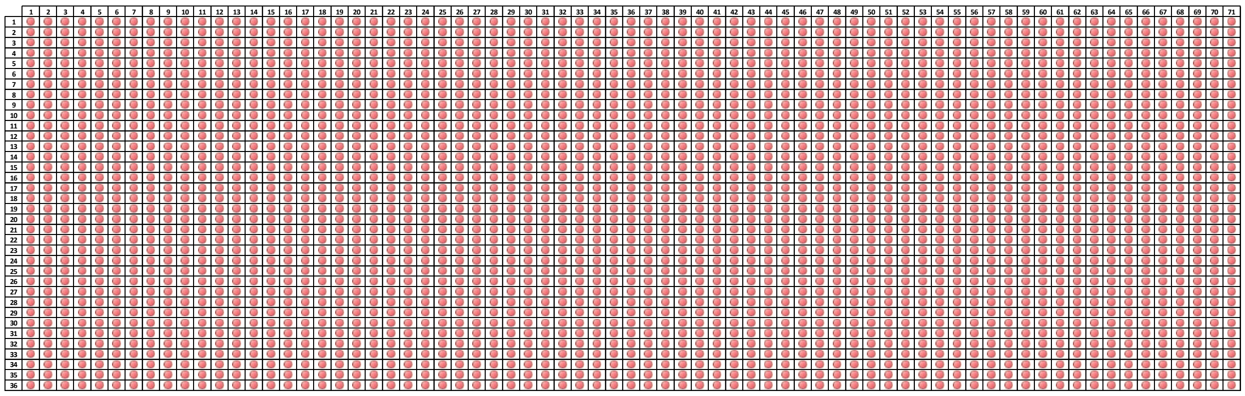


Figure 13: UAV Trajectories in Space/Time (t = 22700 . . . 23700)



Absolute Performance
(Green ≥ 100 visits > Yellow ≥ 75 visits > Red)

Figure 14: Cell Visitation Counts for Controller #3 (Near Simultaneous Visits Removed)

With this in mind, when we control for flocking by removing the near simultaneous visits from the visitation counts, the absolute performance of Controller #3 significantly degrades. Specifically, when contrasted with Figure 8, the decrease in absolute performance shown in Figure 14, is rather astonishing and poor, as the average visitation count drops from 96 to 22.

6 Volume Suppressed

As seen above, metrics 1 - 4 allow us to compare the relative performance of the controllers against each other and a desired standard using a given test case, and, based on the previous section, Controller #3 is inefficient and ineffective. While plotting trajectories in 3-space and examining metrics 1 – 4 are extremely enlightening, it is also time consuming and computationally clumsy. Accordingly, we desire a holistic metric and an automated way to evaluate various UAS parameters (and controllers) to see how efficient and effective a swarm would be given a software solution.

From an operational standpoint, the swarm should limit the ability of the enemy to maneuver freely. With this in mind, we introduce the *enemy's windows of opportunity*, space/time windows for the enemy to maneuver unobserved. In a visual sense, these windows are represented as rectangular prisms within the space/volume absent of any UAS contrails (see Figure 15). That is, over a contiguous portion of the AO throughout a given duration of time, no UAS passed overhead. In this way, the terrain is unobserved, and the enemy may act with impunity. Accordingly, good controllers should minimize the volume of these rectangular prisms. Put another way, they should maximize the space/time *volume suppressed*.

Additionally, there is a minimum amount of time required for an enemy to take action within a given area. For example, if a UAS observed a given grid at 11:30:05 AM and the next UAS passed overhead at 11:30:23 AM, it is unrealistic to think that an insurgent could have placed an improvised explosive device (IED) during the 18 seconds of unobserved time. Moreover, even if the insurgent could place the IED in 18 seconds, he undoubtedly would delay his emplacement both before and after the passage of the first UAS, as he would hear and / or see the UAS both before and after the actual observed time, but he would not know exactly what terrain the UAS was observing. In sum, while there was 18 seconds of unobserved time, the area was successfully suppressed.

In the same way, we need to consider the minimum amount of stand-off distance for an enemy to take action at a given location. Using the example above, if a UAS observed a given grid at 11:30:00 AM and the next UAS passed overhead at 12:30:00 PM, then we might think that any reasonably competent insurgent could have place an IED during the hour of unobserved time. However, if a UAS passed overhead in at least one adjacent grid every 30 seconds during this hour, then it is unrealistic to think that an insurgent would have placed an IED. Simply put, there is just too much UAS activity in close proximity to his location, and, once again, the insurgent does not know exactly what terrain the UASs are observing. The area was successfully suppressed.

```
procedure calculate volume suppressed

  for $x = 1$ to $X$
    for $y = 1$ to $Y$
      for $t = 0$ to $T$
        if there is no UAV in $(x, y, t)$
          then $i = i + 1$
        else
          if $i > t'$
            then $VUS = VUS + i$
          $i = 0$
      end for $t = 0$ to $T$
    end for $y = 1$ to $Y$
  end for $x = 1$ to $X$
  $VS = 1 - VUS/(X \times Y \times T)$
```

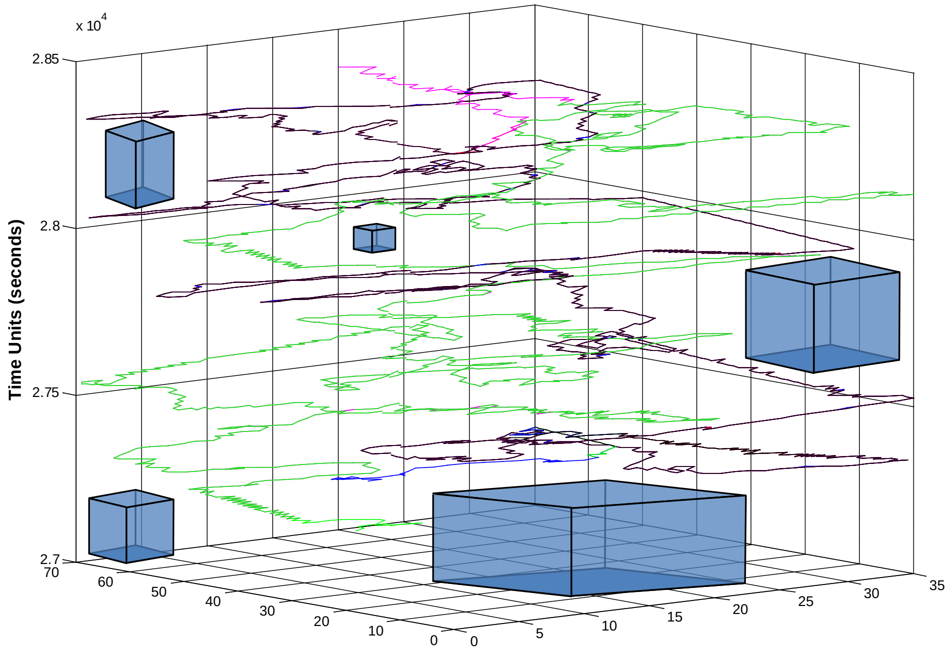


Figure 15: Enemy Windows of Opportunity

With this in mind, we define *width of opportunity* ($w\,'$) and *time of opportunity* ($t\,'$) *thresholds* that establish these minimums. As such, in order for a grid to be successfully suppressed, a UAS need only pass overhead within w' grids and within t' time units of the previous suppression. In the simplest case (e.g., when $w\,' = 0$), the pseudo-code for calculating volume suppressed (VS) from volume unsuppressed (VUS) is given on the previous page.

7 Controller Comparison

Armed with volume suppressed, we are now ready to compare the 9 controllers. Using a swarm of eight UAS ($S = 8$), we set t' $= 25$ time units and record their the performance for w' $= 1$ to 36. As seen in Figure 16, Controllers #2 and #3 clearly underperformed their stochastic competitors, to include the simple random walk.

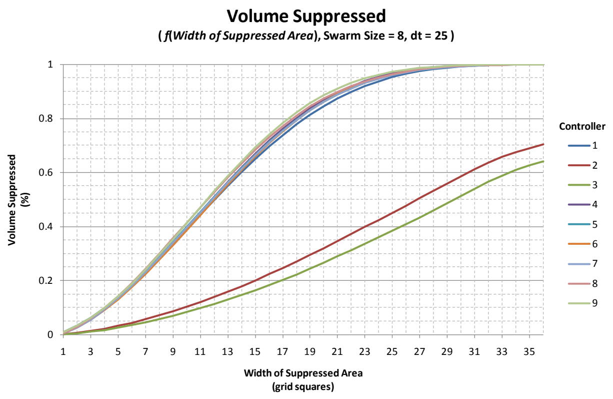


Figure 16: Controller Performance - Volume Suppressed as a Function of Area Width (t' $= 25$)

When t' is increased to 300 time units (5 minutes), we begin to see separation among the stochastic controllers, especially among the mixture models with Controller #7 (a mixture model which randomly selects between Controllers #2 and #3) emerging as the best alternative (see Figure 17). In fact, when t' is increased to 600 time units (representing a 10 minute *time of opportunity* threshold), Controller #7 is able to suppress over 85% of the space/time volume even at an w' of 1 (see Figure 18), while last year's best controller, Controller #3, does not see this performance until w' reaches 24. In short, Controller #7 is very effective.

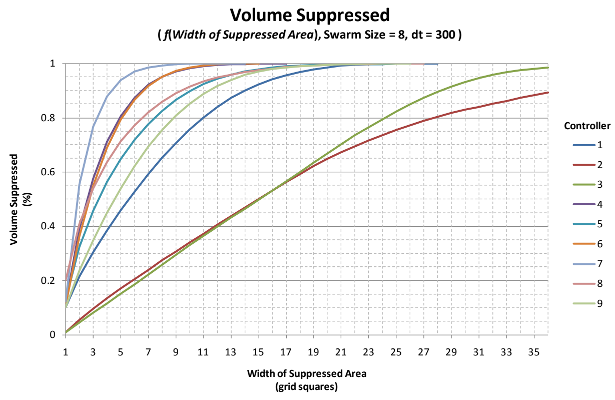


Figure 17: Controller Performance - Volume Suppressed as a Function of Area Width (t' = 300)

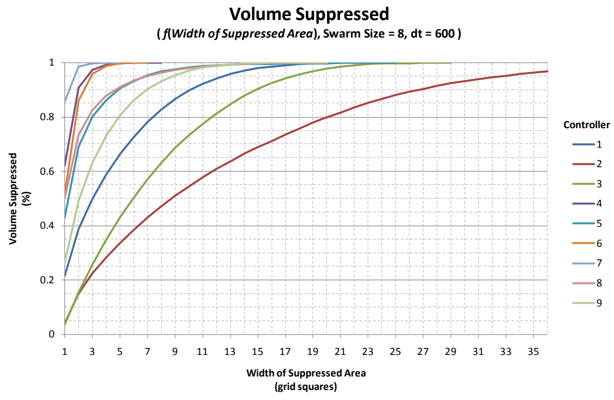


Figure 18: Controller Performance - Volume Suppressed as a Function of Area Width (t' = 600)

While viewing volume suppressed as a function of w' is useful, it is also valuable to see it as a function of the swarm size. That is, given a set t' and w', what is the volume suppressed that can be achieved for a swarm size S? To examine this question, we looked at controller performance for swarms of 1 to 34 with t' and w' set to 300 and 4 respectively. The results of these experiments are given in Figure 19 below.

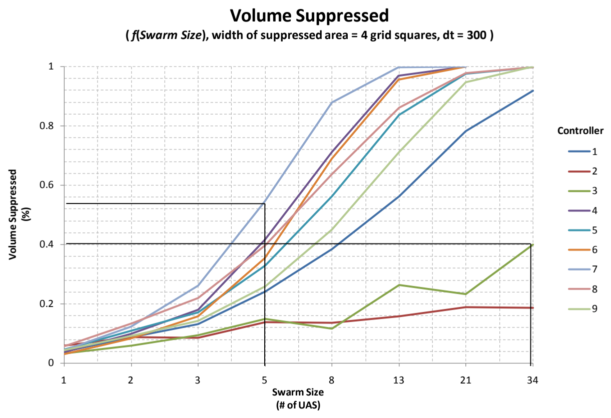


Figure 19: Controller Performance - Volume Suppressed as a Function of Swarm Size

As seen above, Controller #7 outperforms all others for swarms of 3 or more UAS, and, even with a swarm of 34, Controller #3 cannot outperform Controller #7 with a swarm of 5. Simply put, Controller #7 is very efficient.

In light of the above analysis, Controller #7 performs the best in the aggregate. That is, Controller #7 maximizes volume suppressed over the entire space/time volume. Aggregation (or by proportionality averages), however, can be deceiving. With this in mind, we can examine volume suppressed at the individual grid-level to determine where, if at all, Controller #7 may be weak and another controller may be strong.

Accordingly, we set t', w', and S to 300, 4, and 8 respectively, and we compared Controller #7 to Controllers #1, #2, and #3. From Figure 20 on the next page, it is obvious that Controller #7's grid-level volume suppressed (denoted by the light blue markers)

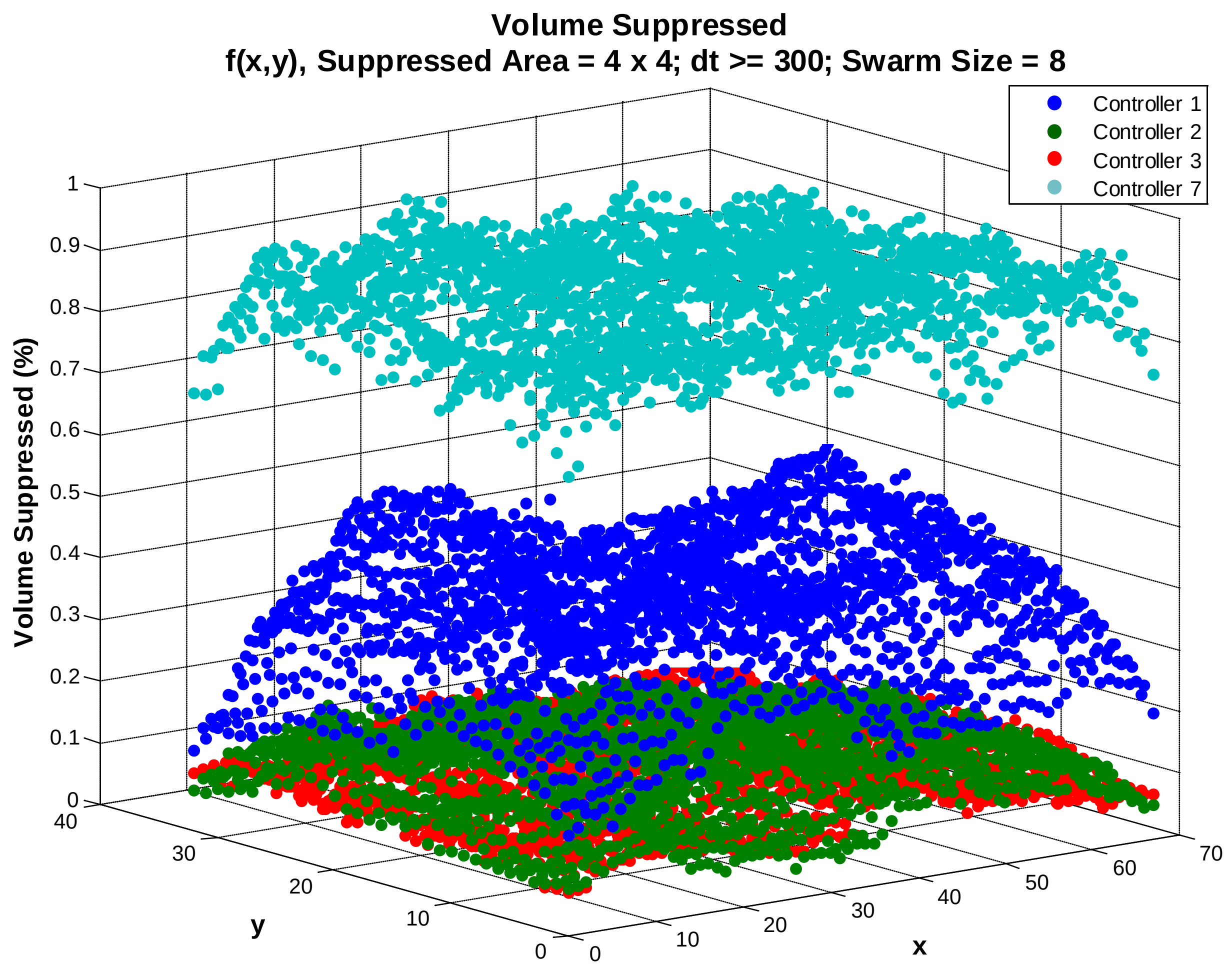


Figure 20: Volume Suppressed as a Function of Location

uniformly outperformed the others. Additionally, it is interesting to note that while Controller #1 (the Symmetric Random Walk) outperformed Controllers #2 and #3; the mixture of Controllers #2 and #3 (Controller #7) was far superior.

8 Future Work

As discussed in Section 2.3, while controlling a swarm of UASs via digital pheromones is currently being pursued by multiple agencies, the research to date has focused on detecting targets that are present and has not focused on area denial. Moreover, documented efforts have neither (1) incorporated intelligence estimates to guide the swarm nor have they (2) used probabilistic updating to influence its behavior. Accordingly, it is natural for our future efforts to take us in these directions.

In the first case, the development of several promising threat assessment methods (namely Riese's Threat Mapper and Huddleston and Browns' Multi-Level Models) make incorporating intelligence estimates relatively straight forward (Karalus & Riese, 2009; Huddleston & Brown, 2009). For example, if we replace the constant decay parameter λ in $p_{x,y}(t) = \gamma_{x,y}e^{-\lambda(t-t_{x,y})}$ with the non-homogeneous, decay parameter, $\lambda_{p_{x,y}} = \lambda(\mathit{1} + \xi(f(\mathit{threat}_{x,y}))$, where (a) $\xi = 1$ if the threat surface is active and 0 otherwise and (b) $f(\mathit{threat}_{x,y}) \in [0,1]$; monotonically increasing, then we can add an appropriate threat premium to cells based on their underlying threat. In theory, this formulation will increase the rate of decay on higher threat cells, causing the UAS to visit them more frequently. This particular approach is attractive, not only due to its simplicity but also its generality, as the threat map can effectively be turned off via ξ.[3]

While this proactive approach uses prior information to influence swarm behavior, the UASs' actual observations provide additional, valuable information, and we should update our threat assessments based on what they observe. With this in mind, we can model the underlying threat probability itself as a random variable with a $Beta(\alpha, \beta)$ distribution.[4] As the Beta distribution is conjugate, if a UAS observes a threat in given cell, then the cell's updated, posterior threat probability distribution is $Beta(\alpha+1, \beta)$, else it is $Beta(\alpha, \beta+1)$. Exploiting this relationship, we can further modify $p_{x,y}(t) = \gamma_{x,y}e^{-\lambda_{p_{x,y}}(t-t_{x,y})}$ as $p_{x,y}(t) = \gamma_{x,y}e^{-(w_1\lambda_{p_{x,y}}+w_2\lambda_{r_{x,y}})(t-t_{x,y})}$ where $\lambda_{r_{x,y}} = \lambda(E(\text{P(threat in (x, y))}))$ and $w_1 + w_2 = 1$. Again, in theory, this will decay cells with higher threat probabilities at a faster rate, thereby focusing the swarm's attention on them (without ignoring the less threatening areas). Moreover, the incorporation of the weighting parameters w_1 and w_2 allows us to tune the model to be more proactive or reactive as required.

9 Conclusions

As mentioned in Section 2.2, the principal research tasks for this study were to:

- Develop improved rule sets for controlling swarming behavior,
- Develop a modeling and simulation test bed for swarming, small UASs in order to test the differential aspects of system components, and
- Evaluate various UAS parameters to see how efficient / effective a swarm would be given a set of hardware (software) and recommend hardware (software) solutions.

Based on the performance of Controller #7, the Haskell simulation given in the Appendix, and the development of volume suppressed (a novel metric for assessing relative controller performance as well as gauging swarm efficacy), we have delivered.

[3]This approach was successfully modeled in 2010 as part of a USMA Department of Systems Engineering cadet capstone project.

[4]Bertuccelli and How use this method for a single UAS in "Robust UAV search for Environments with Imprecise Probability Maps," IEEE, 2005, pg. 5680-5685.

While this is satisfying from an analyst's perspective, it provides little, tangible value if it does not benefit the ultimate stakeholder, the Small Unit Leader. Fortunately, the recent remarks of Dr. Killion, the Chief Scientist of the Army, allay this concern:

> The future operating environment will see a significant increase in the use of unmanned systems requiring the direct employment and monitoring of multiple systems. Unmanned systems must be able to execute complex tactical behaviors in complex environments with minimal required operator control or intervention thus freeing the Soldier / Robotic Controller to be able to accomplish other mission essential tasks such as monitoring the unmanned systems mission package and the tactical situation. Future Force Soldiers must individually possess an expanded ability to control multiple autonomous / semi-autonomous systems (Killion, 2010).

Accordingly, in 20 years time, we may take the "chaotic yet strangely orchestrated" movements of large numbers of UAS for granted, as swarms dutifully collect intelligence, deter enemy action, and perhaps even attack. Surely, its algorithms and technology will make ours look novice and antique by comparison. Nonetheless, should we see such a day as Dr. Killion suggests, then we will know that we played an early albeit small role in a development that helped Soldiers, and that is beyond satisfying – it is everything.

A Haskell Simulation Code

A.1 Main

Main/Run

Cook, J. MR SSD, Inc.

2009

1 Parameters controlling the run

1.1 Types and defaults

```
data RunParameters m aoi = RunParameters
  { runRandomSource :: SomeRandomSource
  , runLength       :: Double
  , runUavFormation :: [(Trajectory, LoiterControllerE)]
  , runHooks        :: RunHooks m
  }
data RunHooks m = RunHooks
  { initializeRun   :: [m ()]
  , initializeUav   :: [UAV m → m ()]
  }
defaultRunParameters = RunParameters
  { runRandomSource = defaultRandomSource
  , runLength       = 30000
  , runUavFormation = take 10
     [(trajectory, LC (RandomWalk (AOI (100, 100)) 1))
      | trajectory ← vFormation 50
     ]
  , runHooks        = defaultRunHooks
  }
defaultRunHooks = RunHooks
  { initializeRun   = []
  , initializeUav   = []
  }
```

1.2 Convenient functions for adding hooks

```
addInitHook hook params = params
  { runHooks = (runHooks params)
```

```
        {initializeRun = hook : initializeRun (runHooks params)
        }
    }
addUavHook hook params = params
    {runHooks = (runHooks params)
        {initializeUav = hook : initializeUav (runHooks params)
        }
    }
addUavEventHandlerHook trigger action
    = addUavHook $ λuav → do
        addUavEventHandler uav trigger action
        return ()
```

2 Invocation of a run

```
doRunWithMMapUpdates cleanup aoi mmap params = do
    doRun $ addUavEventHandlerHook trajectoryChanged
            (scheduleMemoryMapUpdates cleanup aoi mmap) params
doRun RunParameters
    {runRandomSource = randomSource
    , runLength = runLength
    , runUavFormation = uavFormation
    , runHooks = runHooks
    } = runEventGraph $ do
        scheduleEventIn runLength StopSim
        sequence_ (initializeRun runHooks)
        sequence_
            [do
                uav ← mkUav n startTrajectory
                scheduleEventAt (endT startTrajectory) (Arrive uav)
                addUavEventHandler uav trajectoryChanged
                    (doNext ∘ Depart)
                addUavEventHandler uav completedTrajectory
                    (runLoiterController controller randomSource)
                sequence_
                    [initUav uav
                     | initUav ← initializeUav runHooks
                    ]
            | n ← [1 . .]
            | (startTrajectory, controller) ← uavFormation
            ]
```

3 The simulation machinery

Control flow is quite simple, and once you grok it's pretty easy to manipulate. A rough overview of what's going on:

3.1 Arrive

A UAV arrives at its intended destination. It triggers the "completed trajectory" event for the uav, causing the loiter controller to run and select a new trajectory. The assignment of a new trajectory causes the Depart event to be scheduled as well as (possibly many) *UpdateMemoryMapCell* event(s) and callback(s) to *onEnterCell*.

```
data Arrive m = Arrive (UAV m)
instance Monad m ⇒ MonadEvent m (Arrive m) where
  describeEvent (Arrive uav) = do
    uav ← describeUav uav
    return (text "Arrive" <+> parens uav)
  runEvent (Arrive uav) = do
    fireUavEvent uav completedTrajectory uav
    return ()
```

3.2 Loiter controller invocation

Each UAV will have this hook registered to evaluate its loiter controller whenever it reaches the previously assigned destination.

```
runLoiterController randomSrc controller uav = do
  currentPos ← getCurrentPosition uav
  dest ← chooseNextCell randomSrc controller currentPos
  uav `plotTrajectoryTo` dest
```

3.3 Depart

A UAV is leaving a cell. When this event fires, a new *Arrive* event is scheduled at the time the uav is expected to complete its current trajectory.

```
data Depart m = Depart (UAV m)
instance (Monad m, ScheduleEvent m Double (Arrive m)) ⇒ MonadEvent m (Depart m) where
  describeEvent (Depart uav) = do
    uav ← describeUav uav
    return (text "Depart" <+> parens uav)
  runEvent (Depart uav) = do
    trajectory ← getUavTrajectory uav
    now ← getCurrentTime
```

```
if startT trajectory ≉ now
  then fail "Depart fired at weird time"
  else do
    let arrivalTime = max 0 (endT trajectory − now)
    scheduleEventIn arrivalTime (Arrive uav)
    return ()
```

3.4 UpdateMemoryMapCell

A uav is entering a cell. Which uav is irrelevant. The cell will be “touched” and nothing else happens.

```
data UpdateMemoryMapCell mmap cell
   = UpdateMemoryMapCell (TVar mmap) cell Double Double
instance MonadEvent EventM (UpdateMemoryMapCell MMap_ Cell)
  where
    describeEvent (UpdateMemoryMapCell _mmap cell t0 t1) =
      return ∘ fcat ∘ map text
       $  words "Mark memory map at cell"
       ++ [show cell]
       ++ words "at times"
       ++ [show (t0, t1)]
    runEvent (UpdateMemoryMapCell mmap cell t0 t1) =
      modifyReference mmap (addObservationRange cell (t0, t1))
```

3.5 Scheduling updates to the memory map

Whenever a uav’s trajectory changes, *scheduleMemoryMapUpdates* is invoked to schedule *UpdateMemoryMapCell* events for every cell that the specified trajectory will touch, as well as (optionally, controlled by the *cleanup* parameter) a cleanup hook to cancel pending updates if the trajectory changes prematurely.

```
scheduleMemoryMapUpdates cleanup aoi mmap uav = do
  trajectory ← getUavTrajectory uav
  let transitions = cellTransitionsOnTrajectory aoi trajectory
      spans = [(t0, t1, cell)
              | (t0, cell) ← spanEnds
              | (t1, _)    ← drop 1 spanEnds
              ]
      spanEnds =
        maybeToList (timePos2Cell aoi (startT trajectory) (startPos trajectory))
         ++ transitions ++
        maybeToList (timePos2Cell aoi (endT trajectory)  (endPos trajectory))
    -- schedule events and return (time, eventId)
```

```
events ← sequence $
  [do
      -- at time span begins, tag the cell for the duration of the span
    eId ← scheduleEventAt t0 (UpdateMemoryMapCell mmap cell t0 t1)
    return (t0, eId)
  | (t0, t1, cell) ← spans
  ]
  -- add hook to cancel future events if the trajectory changes
when cleanup $ addOneShotUavEventHandler uav trajectoryChanged $ λ_uav → do
  now ← getCurrentTime
  mapM_ cancelEvent
    [event
    | (t, event) ← events
    , t > now
    ]
return ()
```

A.2 Neighborhood

SimpleUav/Neighborhood

Cook, J. MR SSD, Inc.

2009

1 A type describing neighborhoods of cells

At present, only two types of neighborhood are implemented - the empty neighborhood and a very simple neighborhood defined by two radii. The first specifies one half the side length of a square. The second specifies a manhattan distance inside which will not be searched. In all experiments performed for MAJ Dabkowski's work, the values used were 1 and 0 respectively, giving a neighborhood consisting of the 8 cells adjacent to the aircraft's current cell.

```
data Neighborhood a where
  EmptyNeighborhood :: Neighborhood a
  SimpleNeighborhood :: (Num a, Enum a, Ord a, Show a) ⇒
                          a → a → Neighborhood (a, a)
emptyNeighborhood :: Neighborhood a
emptyNeighborhood = EmptyNeighborhood
simpleNeighborhood :: (Num a, Enum a, Ord a, Show a) ⇒ a → a → Neighborhood (a, a)
simpleNeighborhood r1 r2
  | null (cellNeighborhood (SimpleNeighborhood r1 r2) (0, 0))
               = EmptyNeighborhood
  | otherwise = SimpleNeighborhood r1 r2
```

2 Sampling Neighborhoods

The *cellNeighborhood* function converts the specification for a neighborhood into a function which actually lists the cells in the neighborhood of a specified cell. The *CellNeighborhood* type is a convenient alias for the type of such functions.

```
type CellNeighborhood a = a → [a]
cellNeighborhood :: Neighborhood a → CellNeighborhood a
cellNeighborhood EmptyNeighborhood _ = []
cellNeighborhood (SimpleNeighborhood r1 r2) (x, y) =
  [(x + dx, y + dy) | dx ← [−r1 .. r1]
     , dy ← [−r1 .. r1]
```

```
      , abs dx + abs dy > r2
      ]
```

3 Sampling neighborhoods based on position

Similarly, *positionNeighborhood* looks up which cells are in the neighborhood of a specified spatial location. The correspondence additionally depends on an *IrregularCellularAOI* with cells of the appropriate type. Cells that are not a part of the AOI are excluded from the neighborhood, and if none are left after filtering or if the position given is not in the AOI, then a neighborhood is returned consisting of one or more arbitrary cells nearest the position.

```
type PositionNeighborhood a = (Double, Double) → [a]
positionNeighborhood :: IrregularCellularAOI aoi cell ⇒
                        aoi → Neighborhood cell → PositionNeighborhood cell
positionNeighborhood aoi cn position =
  case cellContainingPosition aoi position of
    Nothing  → cellsNearPosition aoi position
    Just cell → let neighborhood =
                        filter (cellInAoi aoi) (cellNeighborhood cn cell)
                in if null neighborhood
                        then cellsNearPosition aoi position
                        else neighborhood
simplePositionNeighborhood ::
    (IrregularCellularAOI aoi (a, a)
    , Num a, Enum a, Ord a, Show a
    ) ⇒ aoi → a → a → PositionNeighborhood (a, a)
simplePositionNeighborhood aoi r1 r2 =
    positionNeighborhood aoi (simpleNeighborhood r1 r2)
```

A.3 Cell Selection

SimpleUav/CellularChooser

Cook, J. MR SSD, Inc.

2009

1 Cell-score-based controller support code

Both the *SimpleNeighborhood* and *QuadrantAveraging* controllers, and many others that might be implemented later, are based on a cell-scoring or -weighting model, where cells are nominated, scored or weighted, and selected based on the score or weight. Two models are implemented here for performing the actual selection for such a scheme.

The *runCellularChooser* uses a cellular chooser to implement the selection part of a controller's task. It produces a list of candidates (which by construction of the *positionNeighborhood* function will not be empty) and chooses from among them according to the cellular chooser. Also by construction of *positionNeighborhood*, the cell chosen will be inside the AOI, so the final conversion back to a cell position will not fail.

```
class CellularChooser chooser score where
  chooseCellByScore :: chooser → [cell]
                    → (cell → score)
                    → RVar cell
data Chooser score where
  Chooser :: (CellularChooser chooser score, Show chooser)
          ⇒ chooser → Chooser score
instance CellularChooser (Chooser score) score where
  chooseCellByScore (Chooser c) = chooseCellByScore c
instance Show (Chooser score) where
  showsPrec p (Chooser c) = showsPrec p c
runCellularChooser
  :: (IrregularCellularAOI aoi cell,
      CellularChooser chooser score)
  ⇒ chooser → aoi → (Double, Double) → Neighborhood cell
  → (cell → score)
  → RVar (Double, Double)
runCellularChooser chooser aoi currentPos neighborhood score
  = do let candidates
             = positionNeighborhood aoi neighborhood currentPos
```

```
    when (null candidates)
      (fail "runCellularChooser: no candidates!")
    winner ← chooseCellByScore chooser candidates score
    case positionOfCell aoi winner of
      Just dest → return dest
      Nothing  → fail ("runCellularChooser: programming error, "
                ++     "selected cell was not in the AOI.")
```

1.1 (Mostly-)Deterministic selection

The *Deterministic* chooser first selects the cell(s) with the highest score. In case of a tie, a uniformly-weighted random selection is made from the list of tied cells.

```
data Deterministic = Deterministic
  deriving (Eq, Show)
instance (Ord score, Distribution Uniform score)
         ⇒ CellularChooser Deterministic score
         where
  chooseCellByScore Deterministic candidates score = do
    let winners = maximaBy (comparing score) candidates
    randomElement winners
```

1.2 Roulette selection

The *Roulette* chooser selects from a list of cells using the cells' scores as relative weights in a discrete distribution.

```
data Roulette = Roulette
  deriving (Eq, Show)
instance (Ord score, Fractional score,
         Distribution Uniform score)
         ⇒ CellularChooser Roulette score
         where
  chooseCellByScore Roulette candidates score
    = discrete scoredCandidates
    where
      scoredCandidates =
        [(score c, c)
        | c ← candidates
        ]
```

A.4 Controllers

SimpleUav/LoiterControllers

Cook, J. MR SSD, Inc.

2009

1 The *LoiterController* class

The *LoiterController* type class specifies the interface loiter controllers must provide. The *LoiterControllerE* type is an existential type allowing loiter controllers of different types to be handled together.

```
class LoiterController c where
  describeLoiterController
      :: c → IO Doc
  chooseNextCell
      :: (MonadIO m, MonadTime m Double, RandomSource m s)
      ⇒ c → s → (Double, Double) → m (Double, Double)
data LoiterControllerE = ∀c.LoiterController c ⇒ LC c
instance LoiterController LoiterControllerE where
  describeLoiterController (LC c) = describeLoiterController c
  chooseNextCell (LC c) = chooseNextCell c
```

2 Some simple controllers

2.1 Random Walk

The *RandomWalk* controller is configured with a distance parameter specifying the maximum number of cells in either dimension it will travel each time it makes a decision. The distance it chooses to travel in each dimension is independent of the other.

If the chosen destination is not inside the AOI, a random cell inside the AOI is chosen from among those nearest the current position of the UAV.

```
data RandomWalk aoi = RandomWalk
  {rwAoi :: aoi
  , rwDist :: Int
  }
instance (IrregularCellularAOI aoi (Int, Int), Pretty aoi)
          ⇒ LoiterController (RandomWalk aoi) where
  describeLoiterController (RandomWalk {..}) =
    return (text "RandomWalk"
      <+> pPrint rwAoi
      <+> pPrint rwDist)
  chooseNextCell (RandomWalk {..}) src currentPos = do
    (dx, dy) ← sampleFrom src (randomJump rwDist)
    let destination = do    -- Maybe monad
          (x, y) ← cellContainingPosition rwAoi currentPos
          positionOfCell rwAoi (x + dx, y + dy)
    case destination of
      Just pos → return pos
      Nothing → sampleFrom src (jumpIntoAoi rwAoi currentPos)
randomJump dist = jump
  where
    jump = do
      dx ← uniform (−dist) dist
      dy ← uniform (−dist) dist
      if dx ≡ 0 ∧ dy ≡ 0
        then jump
        else return (dx, dy)
jumpIntoAoi aoi currentPos = randomElement candidates
  where candidates = cellPositionsNearPosition aoi currentPos
```

2.2 Local Neighborhood

The *SimpleNeighborhood* controller implements the simple exponential-decay pheremone-based local neighborhood controller. Operating within the specified AOI, according to the specified cellular chooser, it selects a cell from within the specified neighborhood based on scores derived from the memory map.

The score of a cell is given as one minus the pheremone level for that cell. When operating under the deterministic chooser, the cell chosen will be the one with the maximum score (and therefore the minimum pheremone level). When using the roulette chooser, the score will be interpreted as a relative weight for that cell.

```
data SimpleNeighborhood aoi chooser cell mmap
  = SimpleNeighborhood
  {snAoi           :: aoi
  , snChooser      :: chooser
  , snLambda       :: Double
  , snNeighborhood :: Neighborhood cell
  , snMMap         :: TVar mmap
  }
instance (MemoryMap mmap cell
, IrregularCellularAOI aoi cell
, Pretty aoi
, Ord cell, Show cell
, CellularChooser chooser Double
, Show chooser
) ⇒ LoiterController
(SimpleNeighborhood aoi chooser cell mmap)
where
  describeLoiterController (SimpleNeighborhood {..})
    = return (text "SimpleNeighborhood:"
      <+>   pPrint snAoi
      <+>   text (show snChooser)
      <+>   pPrint snNeighborhood)
  chooseNextCell (SimpleNeighborhood {..}) src currentPos
    = do
        mmap ← readReference snMMap
        t ← getCurrentTime
        let score cell = 1 − readDecayLevel snLambda mmap cell t
        let destination = runCellularChooser snChooser snAoi
                              currentPos snNeighborhood score
        sampleFrom src destination
```

3 Composite controllers

3.1 "Mixed" controller

Each invocation of *MixedController* randomly chooses from the provided list of controllers with the associated relative probability weights.

```
newtype MixedController
  = MixedController [(Double, LoiterControllerE)]
instance LoiterController MixedController where
  describeLoiterController (MixedController cs) = do
    descriptions ← sequence
      [do
        descr ← describeLoiterController c
        return (hang empty 8 (pPrint p <> colon) <+> descr)
      | (p, c) ← cs
      ]
    return (text "MixedController:" <+> fcat descriptions)
  chooseNextCell (MixedController cs) src currentPos = do
    c ← sampleFrom src (discrete cs)
    chooseNextCell c src currentPos
```

A.5 Output Analysis

Main/Analysis

Cook, J. MR SSD, Inc.

2009

Abstract

This module defines and implements the basic administrative machinery used to specify which analysis data to generate and where to send it.

1 Data types for controlling run analysis output

The *AnalysisProduct* type is an algebra for specifying a set of data to be collected during or after a run. The *AnalysisParams* type defines all user-tuneable parameters controlling the data collection and analysis for a run, including a definition of where (if anywhere) each analysis product should be sent.

The *anaOutput* field of *AnalysisParams* specifies zero or more destinations for each *AnalysisProduct* to be generated. Its type is parameterized because it will initially be specified as a *String* specifying the file path, but before the run those files will be opened and their paths replaced by file handles.

```
data AnalysisProduct
   = Metadata
   | MMapUpdates
   | MMapUpdatesByUav Int
   | MMapUpdatesByCell Cell
   | MMapCoverage
   | VoidsBySquare Square
   | VoidsByCell Cell
   | VoidsBySize Int
   | VoidLengthsBySquare
   | VoidLengthsBySize Int
   | MaxExtentBySquare
   | ExtentStatsBySquare
   | ExtentStatsBySize
  deriving (Eq, Ord, Read, Show)
data AnalysisParams h = AnalysisParams
  {anaRunName          :: String
  , anaLoggerName      :: String
  , anaOutput          :: Map.Map AnalysisProduct [h]
  , anaUnobsFilter     :: Int → Double → Bool
  , anaAoi             :: AOI
  , anaSeqMMap         :: Bool
  , anaCleanupEvents   :: Bool
  }
```

2 Predefined sets of parameters

Functions to construct analysis parameters describing standard sets of analysis outputs. These are parameterized by run names, because they specify default filenames for the outputs as well as which outputs to produce.

```
defaultAnalysisOutputs runName = Map.fromList
  [(Metadata,       [runName ++ ".txt"])
  , (MMapCoverage,  [runName ++ ".txt"])
  , (MMapUpdates,   [runName ++ "_mmap.txt"])
  ]
minimalAnalysisOutputs runName = Map.fromList
  [(Metadata,       [runName ++ ".txt"])
  , (MMapCoverage,  [runName ++ ".txt"])
  ]
defaultAnalysisParameters runName = AnalysisParams
  {anaRunName        = runName
  , anaLoggerName    = runName ++ ".analysis"
  , anaOutput        = defaultAnalysisOutputs runName
  , anaUnobsFilter   = λ_sz _dt → True
  , anaAoi           = AOI (100, 100)
  , anaSeqMMap       = True
  , anaCleanupEvents = False
  }
```

3 Output file management

Functions to handle opening and closing of the analysis output files used in a run. *openAnalysisFiles* is slightly more complex than might be expected, because it caches the open file handles to avoid opening the same file more than once if it is the destination for multiple analysis products.

```
openAnalysisFiles anaParams@AnalysisParams
  {anaOutput = outputNames} = do
    let logger = anaLoggerName anaParams
```

```
        ++ ".openAnalysisFiles"
    files ← newRef Map.empty
    openedOutputs ← Traversable.forM outputNames $ λnames →
      forM names $ λname → do
        cached ← readsRef files (Map.lookup name)
        case cached of
          Just handle → do
            debugM logger $ unwords
              [name
              , "already opened, returning cached handle"
              , show handle
              ]
            return handle
          Nothing → do
            debugM logger $ unwords
              [name
              , "not yet opened, opening and caching"
              ]
            handle ← openFile name WriteMode
            hSetBuffering handle (BlockBuffering Nothing)
            modifyReference files (Map.insert name handle)
            return handle
    return anaParams {anaOutput = openedOutputs}
closeAnalysisFiles AnalysisParams {anaOutput = outputFiles}
  = sequence_
    [do
      isOpen ← hIsOpen handle
      when isOpen $ do
        hFlush handle
        hClose handle
    | handles ← Map.elems outputFiles
    , handle ← handles
    ]
```

4 Analysis output routing

The following functions manage the routing and formatting of the analysis output data. *lookupOutput*, *lookupOutputs* and *lookupOutputWhere* retrieve the file handles to which a particular piece of analysis data should be printed.

The various *output*... functions compute and format the data and send it to the appropriate files. The *multiOutput*... functions do the same for data considered to be a part of multiple analysis products.

Haskell's lazy evaluation is used in a critical way here - generally speaking, these functions are called for all analysis products, regardless of whether they will be output (or of how many times they will be output). The analysis data will be passed to these functions in unevaluated form, so that if it is not formatted and printed, it will not be unnecessarily computed. Similarly, if it is printed more than once, it will not be unnecessarily recomputed or reformatted.

```
lookupOutput prod anaParams =
  Map.findWithDefault [] prod (anaOutput anaParams)
lookupOutputs prods =
  lookupOutputsWhere (∈ prods)
lookupOutputsWhere p anaParams = nub
  [file
  | (prod, files) ← Map.toList (anaOutput anaParams)
  , p prod
  , file ← files
  ]
 {-# INLINE outputLn #-}
outputLn anaParams outType line
  | null outputs = return ()
  | otherwise    = liftIO $ sequence_
    [hPutStrLn out line
    | out ← outputs
    ]
  where outputs = lookupOutput outType anaParams
outputRow anaParams outType row
  | null outputs = return ()
  | otherwise    = liftIO $ sequence_
    [hPutStrLn out (tabsep row)
    | out ← outputs
    ]
  where outputs = lookupOutput outType anaParams
outputRowWhere p anaParams row
  | null outputs = return ()
  | otherwise = liftIO $ sequence_
    [hPutStrLn out ∘ tabsep $ row
    | out ← outputs
    ]
  where outputs = lookupOutputsWhere p anaParams
multiOutputRow anaParams outTypes row
  | null outputs = return ()
  | otherwise    = liftIO $ sequence_
      [hPutStrLn out (tabsep row)
      | out ← outputs
      ]
  where outputs = lookupOutputs outTypes anaParams
```

```
outputTable anaParams outType table
  | null outputs = return ()
  | otherwise    = liftIO $ do
    infoM logger $ unwords
      ["Outputting table for"
      , show outType
      , "to"
      , show (length outputs)
      , case outputs of
        [_] → "file."
        _   → "files."
      ]
    dt ← time_ $ sequence_
      [hPutStrLn out (tabsep row)
       | row ← table
      , out ← outputs
      ]
    infoM logger $ unwords
      ["CPU time used:"
      , show dt
      , "seconds"
      ]
  where
    outputs = lookupOutput outType anaParams
    logger = anaLoggerName anaParams ++ "." ++ show outType
multiOutputTable logger anaParams outTypes table
  | null outputs = return ()
  | otherwise    = liftIO $ do
    debugM logger $ unwords
      ["Outputting table for"
      , show outTypes
      , "to"
      , show (length outputs)
      , case outputs of
        [_] → "file."
        _   → "files."
      ]
    dt ← time_ $ sequence_
      [hPutStrLn out (tabsep row)
       | row ← table
      , out ← outputs
      ]
    debugM logger $ unwords
      ["CPU time used:"
      , show dt
    ,"seconds"
    ]
where
  outputs = lookupOutputs outTypes anaParams
```

References

Abatti, James M. 2005. *Small Power: The Role of Micro and Small UAVs in the Future.* M.Phil. thesis, Air Command and Staff College.

Arquilla, John, & Ronfeldt, David. 2000. *Swarming and the Future of Conflict.* RAND Corporation.

Bae, Youngchul. 2004. Target Searching Method In The Chaotic UAV. *IEEE*, 12.D.8.1–12.D.8.9.

Bertuccelli, L. F., & How, J. P. 2005. Robust UAV Search for Environments with Imprecise Probability Maps. *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference 2005*, 5680–5685.

Dasgupta, P. 2008. A Multi-agent Swarming System for Distributed Automatic Target Recognition. *IEEE Transactions on Systems, Man, Cybernetics*, **38**, 549–563.

Department of Defense. 2009 (April). *FY2009-2034 Unmanned Systems Integrated Roadmap.*

Department of the Army. 2006. *FMI 3-04.155: Army Unmanned Aircraft System Operations.*

Huddleston, Samuel H., & Brown, Donald E. 2009. A Statistical Threat Assessment. *IEEE: Transactions On Systems, Man, and Cybernetics*, **39**, 1307–1315.

Karalus, Randall, & Riese, Stephen R. 2009 (February). *Threat Mapper Orientation.*

Killion, Thomas. 2010 (May). *10 Comprehensive Warfighter Outcomes.*

Parunak, H., Purcell, M., & OConnell, R. 2002. Digital Pheromones For Autonomous Coordination Of Swarming UAV's. *AIAA*, 1–9.

Shachtman, Noah. 2008 (March). *Deadly Drone Shortage in Iraq.*

Shachtman, Noah. 2009 (February). *Drone Surge.*

Sujit, P.B., Sinha A., & Ghose, D. 2006. Multiple UAV Task Allocation Using Negotiation. *AAMAS*, 471–478.

Teague, Edward, & Kewley, Robert. H. 2008. ORCEN Technical Report: DSE-TR-0808 - Swarming Unmanned Aircraft Systems. *DTIC: ADA489366*, 1–30.

Walter, B., Sannier, A., Reiners, D., , & Oliver, J. 2006. UAV Swarm Control: Calculating Digital Pheromone Fields with the GPU. *JDMS*, **3**, 167–176.

Nomenclature

AMRDEC	Aviation and Missile Research, Development, and Engineering Center
IED	Improvised Explosive Device
ORCEN	Operations Research Center of Excellence
OSD	Office of the Secretary of Defense
SASC	Semi-Autonomous Self-Organizing
UAS	Unmanned Aerial Systems